Scientific Programming with Python: part I
Advanced Research Computing
Outline

• A tour of Python
  – Date types, exceptions
  – Functions, modules, generators
  – Functional programming
  – The Map-reduce data-parallel pattern

• Linear algebra in Python
  – Saxpy as a map operation
  – Using C extensions from Python

• Data-parallel Python
A TOUR OF PYTHON
What is Python?

• Python is a high-level, multi-paradigm, scripting language
• Scripting languages trade performance for productivity
  – dynamic typing improves productivity
  – Interpreted nature supports interactive development
• Multi-paradigm
  – Procedural programming: uni- and multiprocessing
  – Functional programming: data parallel primitives
  – Object oriented programming
Python interpreters

• Two interpreters are widely used
  – Python
  – IPython
• Both support an interactive mode (also know as the Python shell)
  – the user enters Python statements and expressions that are evaluated as they are entered
  – improves productivity
• Both support the help() function
• IPython has a sophisticated line completion features and context sensitive help
Online Python Learning Tool

- An excellent Python learning site with visualization of program execution
  
  http://people.csail.mit.edu/pgbovine/python/

- We will look in class at some examples
  - data types, exceptions
  - factorial function, map, and lambda functions

- Exploring the learning tool on your own after class is recommended
Tuples

- Tuple: an immutable, heterogeneous sequence
  - A comma-separated list of values
  - The values can be of any type and are indexed starting from zero
  - Slices can be extracted using indexing

```python
t1 = (); t2 = (1, );
t3 = tuple('hello'); t3[0]; t3[-1]
a=1, b=2; a,b = (b, a)
l3 = list(t3)
```
Lists

• Lists are mutable, heterogeneous sequences

```python
l0 = [];
l1 = [1, 2, 3.14, 9.81]; l2 = range(4);
l1.index(3.14); l1.count(2)
del l1[3]; l1.remove(3.14); l1.append(4.5)
# okay for list, but not for tuple
l1[2] = 3; l1[2:] = (3,4)
pairs = zip(l1, l2)
l1_l2 = l1 + l2; l2x2 = l2 * 2
l3 = list(t3); l3[0]; l3[-1]; l3[1:]
len(l3); type(l3)
```
Sets

- Sets are unordered collections of immutable objects (ints, floats, strings, tuples)

```
s1 = set([1, 2, 3, 4])
s1.add(6); s1.add(5)

s2 = set([1, 2, 3, 4, 5, 4,3,2,1])
s1 | s2; s1 & s2; s1 ^ s2

s3 = frozenset( set([5]) )
s1.add(s3)
```
Dictionaries

- Python’s associative arrays

```python
e2f= {}
e2f = { ‘blue’: ‘bleu’, ‘red’: rouge}
e2f[‘green’] = ‘vert’
e2f[‘blue’]; e2f.get(‘blue’)
‘red’ in e2f; del e2f[‘green’]

list(e2f.keys()); list(e2f.values());
list(e2f.items());  len(e2f)
num = { 1: ‘one’, 2: ‘two’}; num1 =num.copy()
```
Iterables

• An iterable is an object that has an iterator
  – The iterator is obtained with the `__iter__()` method or with the `iter()` function
  – can be iterated through with the construct:
    ```python
    for item in iterable
    ```

• Lists and tuples are iterable objects, but there are others
  – NumPy arrays
  – Generator objects

• Some functions (e.g., `sum`) require an iterable
Control Flow

• If, for, and while

```python
x = [ -1, 2, 3] ; i = 0
for el in x:
    if x < 0:
        print ‘negative’
    elif x== 0:
        print ‘zero’
    else:
        print ‘positive’
while (i < len(x)):
    print x[i]; i += 1
```
Functions

• Functions have dynamically typed arguments
• The number of function arguments can be fixed or variable
  – The function definition indicates whether it accepts a variable number of arguments
• Recursive vs iterative function implementation
  – Cost of function call + if statement vs cost of for loop
  – Example the factorial function
Factorial: recursive version

• Recur until the base case n<= 1 is reached

```python
def factorial_rec(n):
    if (n <= 1):
        return 1
    else:
        return n * factorial_rec(n - 1)

print factorial_rec(4)
```
Factorial: iterative version

• Compute iteratively the factorial function for 1, 2, …, n using a while loop over 1, 2, …, n.

```python
def factorial_iter(n):
    res = 1
    i = 1
    while (i <= n):
        res *= i
        i += 1
    return res

print factorial_iter(4)
```
Mutable Objects as Arguments

• Arguments are passed by object reference
  – the parameter becomes a new reference to the object

• Passing an immutable object (number, string, tuple) means that the function cannot modify the object

• Passing a mutable object (list, dictionary) means that any change made by the function to the corresponding parameter will change the argument in the caller’s environment
Function with variable number of parameters

- The unknown number of arguments is indicated by *args

```python
def show_list(list, *args):
    last = len(list); step = 1
    if (len(args) >= 1):
        last = args[0]
    if (len(args) >= 2):
        step = args[1]
    print 'stop_idx =', last, 'step =', step
    return [list[i] for i in range(0,last,step)]

list = range(8)
show_list(list); show_list(list, 4); show_list(list, 4, 2)
```
Generator Functions

• A generator is a special function that contains one or more yield statements
  – when called, it returns a generator object
  – the object is iterable: calling next() on the generator object executes the function up to and including the next yield statement

• Generators implement state machines
  – Functions that are not generators should be stateless
Modules

• Are python files (possibly in directory tree) that are loaded using the import statement
  – import loads the module namespace, compiles and executes the module
  – a namespace is am map from names to objects
  – different ways to import allow to control the namespace loaded
Importing a Module

• Access the entire module namespace prefixed with the module name:
  
  ```python
  import module_name
  ```

• Access one name from the module namespace, without prefix
  
  ```python
  from module_name import other_name
  ```

• Access all names from the module namespace, without prefix
  
  ```python
  from module_name import *
  ```
  – must be used with caution to avoid name collisions
Generator for a finite sequence

- If `next()` is called after the last `yield` has been executed, a `StopIteration` exception is generated.

```python
def hello_gen():
    yield 'hello'
    yield 'world'

hello = hello_gen()
list(hello)
hello = hello_gen()
while 1:
    try:
        hello.next()
    except StopIteration:
        print 'DONE'
    break
```
Generator for an infinite series

• Generate 1-1/3+1/5 ... which converges to pi/4

```python
def pi_series():
    sum = 0.0; i = 1.0; j = 1
    while(1):
        sum = sum + j/i
        yield 4*sum
        i = i + 2; j = j * -1
series = pi_series()
pi = 0.0; n = 0
for elem in series:
    n += 1
    if abs( elem - pi ) < 0.001:
        print "PI =", pi, "for", n, "terms in the series 1 - 1/3 + 1/5 - ..."
        break
    else:
        pi = elem
```
Exceptions (1)

- Python, like Java and C++, indicates that an error has occurred in the execution by raising an exception rather than be setting an error code that contains one or more yield statements
  - exceptions are typed objects and they can be caught by type
  - example exception types: ZeroDivisionError, TypeError
Exceptions (2)

• Exception types caught are listed from specific to general

```python
def reciprocals(n):
    try:
        print "reciprocal of ", n, " is", 1./n
        reciprocals(n-1)
    except ZeroDivisionError:
        print "ZeroDivisionError for reciprocal of n =", n
    except:
        print "Error in computing reciprocal of n =", n

reciprocals(3)
```
Classes

• Every class has the `__init__()` method - constructor
• Every method has as first parameter `self`, which is a reference to the class instance
• Private instance variables can be accessed only from inside the object

```python
class Circle:
    def __init__(self, radius=1):
        self.__radius = radius
    def get_radius(self):
        return self.__radius
    def set_radius(self, radius):
        self.__radius = radius
```
Functional Programming (1)

• Imperative programming
  – Program seen as a sequence of computation steps
  – Focus is on *how to compute* the solution of the problem

• Functional programming
  – Program seen as evaluations of mathematical functions
  – Focus is on *what to compute* to solve the problem
Functional Programming (2)

• Functions are first class objects
  – Closure: functions can access variables in the referencing environment
  – No side effects: (pure) functions do not store data in memory, just return the results
    • Stateless programs

• Lambda functions are anonymous functions
  \[\text{lambda } i : i \% 2 == 0\]

• Using functions instead of imperative constructs (e.g., \texttt{for} loops) enables parallel implementation
Functional Programming (3)

• Python is a multi-paradigm language, not a functional language
• Python includes some of the features of functional languages
  – Passing functions to other functions
  – Lambda functions
  – A set of (pure) functions: map(), sum(), all(), any()
Map function

• Apply a function `func` to all the elements of a sequence: `map(func, iterable)`

• Example: convert the elements of a list from integer to float

  `L1 = [1, 2, 3, 4]`

  # using a for loop
  `L2 = [ float(elem) for elem in L1 ]`

  # using a map
  `L3 = map(float, L1)`
Map: recursive version

• Recur until an empty list is reached

```python
def map_rec(func, lst):
    if lst == []:
        return []
    else:
        return [func(lst[0])] + map_rec(func, lst[1:])
```

```python
input = [-2, -4, 6]
print input
output = map_rec(abs, input)
print output
```
Map: iterative version

• Evaluate the function for all the elements of the list using a for loop over the list

```python
def map_iter(func, lst):
    return [func(item) for item in lst]
```

```python
input = [-2, -4, 6]
print input
output = map_iter(abs, input)
print output
```
Reduce function

• Reduce function

\[
\text{reduce ( function, iterable, initializer=None)}
\]

• Converts an iterable to one value

• Sum as a reduction operation:

```python
def sum_red(seq):
    reduce ( lambda x,y: x+y,  seq)

x = range(10)
print sum_red(x)
```

• Sum can also be cast as a \textit{map-reduce} operation
Map-Reduce Example (1)

• Consider the problem of determining whether all the elements of a list are even numbers

• Imperative programming solution uses a function that
  – Initializes the partial result to True
  – iterates over the elements of the list and computes a flag that is True if the element is even and False otherwise
  – Computes the logical AND between the flag and the partial result
  – Returns the results after the list has been traversed
Map-reduce Example (2)

• Imperative style of determining whether the elements of a list are even numbers

```python
def even_iter( iterable ):
    res = True
    for item in iterable:
        res = res and (item % 2 == 0)
    return res

even_iter( [ 2, 4, 6] )
```

• The `for` loop imposes an unnecessary order of operations
  – Goal: express the computation in a way that does not impose an unnecessary order
Map-reduce Example (3)

• Use the map-reduce pattern
  – Map: compute for all items (item%2==0)
  – Reduce: call all() on the list built with map()

```python
even = lambda item: item %2 == 0
def even_map_red( iterable ):
    return all ( map (even, iterable) )
even_map_red( [ 2, 4, 6] )
```

• No unnecessary ordering of operations
  – Enables data-parallel implementation using frameworks such as Copperhead (UC Berkeley)
Sum using map-reduce (1)

- Sum as a reduce operation

- Sum as a map-reduce operation
LINEAR ALGEBRA IN PYTHON
Using BLAS from Python

• BLAS can be used either:
  – Directly from python, as an external module
  – Via a Python package, `scipy.linalg.blas`

• First, we look at invoking BLAS directly from Python
  – We can invoke any BLAS implementation that is packaged as a shared library (.so) file
  – We will use Intel Math Kernel Libraries (MKL) that include BLAS
BLAS Functions

• BLAS function prefix
  – Real: s = single precision, d = double precision
  – Complex: c = single precision, z = double precision

• BLAS1: vector operations
  – ddot  dot product of two vectors: r = x y
  – daxpy scaled vector sum: y = a x + y

• BLAS2: matrix vector operations
  – dgemv matrix vector product

• BLAS3: matrix matrix operations
  – dgemm = matrix-matrix multiplication
MKL BLAS Functions

• FORTRAN functions prototypes are defined in
  – mkl_blas.fi  for Fortran 77
  – blas.f90 , blas95.mode  for Fortran 95

• C prototypes defined in
  – For Fortran 77 BLAS : mkl_blas.h
  – For CBLAS: mkl_cblas.h
  – mkl_blas.h and mkl_cblas.h are  included by mkl.h

• When calling BLAS from C, note that Fortran assumes column-major order of matrix elements
The CBLAS saxpy function

• Prototype of the C saxpy function

```c
void cblas_saxpy(
    const MKL_INT N,
    const float alpha,
    const float * X, const MKL_INT incX,
    const float * Y, const MKL_INT incY);
```
Calling BLAS saxpy from Python

• Pass to the saxpy function the correct ctypes

```python
from ctypes import *

mkl = cdll.LoadLibrary("libmkl_rt.so")
cblas_saxpy = mkl.cblas_saxpy

n = 8; alpha = 1.0
xp = c_float * n;
yp = c_float * n
x = xp(0.0); y = yp(0.0)
for i in range(len(x)):
    x[i] = 1.0; y[i] = 1.0
cblas_saxpy( c_int(n), c_float(1.0), 
              byref(x), c_int(1), byref(y), c_int(1))
```
BLAS and MKL links

• Intel MKL page

• Intel MKL User Guide

• Explore the BLAS functions
  • [http://www.netlib.org/lapack/explore-html](http://www.netlib.org/lapack/explore-html)
Calling SciPy saxpy from Python

- Pass to the saxpy function the types expected by the Python function

```python
import scipy.linalg
import numpy as np

n = 8; a = 1.0
x = np.ones(n, 'f')
y = np.ones(n, 'f')

# use print(saxpy.__doc__) to get the signature
y = scipy.linalg.cblas.saxpy
z = saxpy(a, x, y)
```
DATA-PARALLEL PYTHON
SAXPY as a map operation

• Data parallelism

\[ x_i \quad y_i \]

a*x_i + y_i

• The specification of saxpy must not impose an unnecessary order of operations
Imperative specification of saxpy

The following code inhibits parallelization because

– Parallelization requires a priori knowledge of the sequence `range(len(y))`
– The function has side effects: it changes `y`

```python
def saxpy(a, x, y):
    for i in range(len(y)):
        y[i] = a*x[i] + y[i]
    return y
```
Declarative specification of saxpy

The code below enables parallelization

- No side effects, no unnecessary order, no unknown indices
- Closure: lambda gets a from caller’s environment

```python
def saxpy(a, x, y):
    # or return [ a*xi + yi for xi,yi in zip(x,y) ]
    return map(lambda xi, yi: a*xi + yi, x, y)
```
Relaxing the order

- Python orders the computation sequentially
  - top to bottom of statements
  - blas.f90, left to right `for i in iterable:`
  - inside to outside and left to right of expressions

- Relaxing this order enables to
  - evaluate expressions out order
  - execute map() in arbitrary order

- Annotated saxpy in Python
Annotated saxpy

The annotation @cu indicates that this is CopperHead code that will be compiled to a GPU executable using Just In Time Specialization (JITS)

```python
@cu
def saxpy(a, x, y):
    return map(lambda xi, yi: a*xi + yi, x, y)
res_gpu = saxpy(2.0, x, y)
res_cpu = saxpy(2.0, x, y, cuEntry=False)
```
Productivity AND efficiency

- Data-parallel Python, e.g., Copperhead, combines productivity and efficiency
Thank you.

Questions?